# Linked Data Science Powered by Knowledge Graphs: A Literature Review Report

Mossad Helali

Report submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

July 3, 2023

**Abstract**

With the recent growing interest in data science from both academia and enterprise, a large amount of data science artifacts such as datasets and associated data science pipeline scripts are collected and openly shared among data scientists. Yet, existing research has focused on studying datasets or pipeline scripts in isolation; there has so far been no systematic attempt to holistically exploit the knowledge captured in both artifacts such as compatible datasets, data cleaning steps, and suitable machine learning algorithms. In this report, a literature review is conducted on research work pertaining to two key research areas: dataset discovery in data lakes and semantic code understanding. Dataset discovery is the problem of finding related datasets among a large collection of datasets. Semantic code understanding is the problem of extracting the semantics of code scripts and representing them in a suitable data structure. Understanding prior research work in these areas paves the way to building a linked data science platform, where the semantics of datasets and associated pipelines are collected, captured in a universal representation, and utilized to build novel applications.

# Table of Contents

# 1 Introduction

Data science is the process of collecting, cleaning, and analyzing structured and unstructured data to build predictive models for particular or gain insights pertaining to specific business domains. Data science is founded on two primary elements: the collected datasets and the developed code built on top of them. In recent years, there has been a growing interest in using data science methods and applications. This interest is not just from academics but also from enterprise practitioners, who invest a significant amount of time collecting, organizing, and sharing data and code to automate various business tasks. Furthermore, open data science and collaborative portals, such as Kaggle [26] and OpenML [49], are expanding in popularity as well; the Kaggle portal, for instance, contains tens of thousands of open datasets and hundreds of thousands of associated data science pipelines [36]. Despite the importance of efforts and their large-scale nature, existing systems do not offer a holistic approach in collecting and analyzing data science artifacts such as datasets and pipeline scripts. Instead, they consider these artifacts only in isolation from each other; there is little or no support to help users learn from the experiences and best practices connecting datasets and pipelines.

A data scientist building a pipeline, for instance, is generally interested in datasets relevant to the task at hand, whether on open data portals or within their enterprise, as well as in previously built pipelines using these datasets. Exploring relevant datasets is a core task to perform e.g. data enrichment (by having more rows or columns) and data integrity checks. The associated data science pipelines then allow the data scientist to benefit from the accumulated (public or enterprise) domain knowledge about the datasets, which helps accelerate the development and discovery of new solutions and possibilities. However, existing platforms offer only isolated and limited support for dataset search or pipeline exploration.

In this report, a literature review is conducted on research work pertaining to the areas of data discovery and semantic code understanding. The objective is to understand the existing techniques and their limitations in order to develop a *linked data science platform*: a platform that integrates various data science

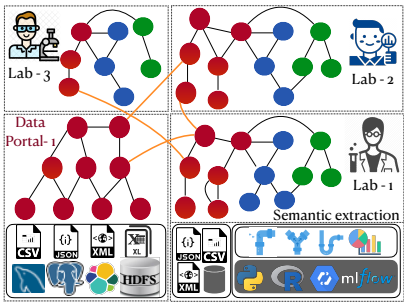



Figure 1: A vision of linked data science platforms, as proposed by, [34]. Linked data science platforms break down the silos between data science labs.

artifacts holistically in a single representation as shown in Figure 1. Linked data science platforms will facilitate the exploration of different data science artifacts, the automation of tasks that utilize them, and sharing of the gained insights among data scientists.

This report is structured as follows: Section 2 pertains to the problem of data discovery, explaining the taxonomy of related work (Section 2.1), followed by an illustration of example data discovery systems (Section 2.2), and a description of data discovery evaluation settings (Section 2.3). Next, Section 3 follows a similar structure for the problem of semantic code understanding. Finally, open research challenges are highlighted in Section 4 and Section 5 concludes.

# 2 Data Discovery

With the growing success of data science, thousands of machine-readable datasets are collected by data owners to form what is commonly known as data lakes. A data lake is a large collection of structured, semi-structured, or unstructured datasets possibly originating from a multitude of data sources as shown in Figure 2.

One of the primary challenges in working with data lakes is the issue of data discovery. Because datasets are stored without any predefined structure or schema,

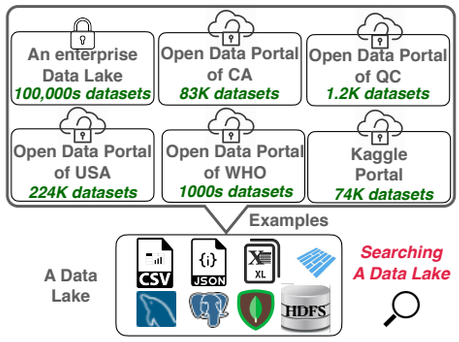



Figure 2: Examples of Data Lakes

it is often challenging for data scientists to manually identify relevant datasets to their data science tasks within a large collection of datasets. As a result, they might rely on the knowledge of their colleagues in an enterprise or seek commonly available knowledge on the web. However, this entails spending an extensive amount of time and effort and does not guarantee finding the relevant datasets. To address this challenge, researchers have developed various techniques for dataset discovery in data lakes. These techniques aim to identify and categorize datasets within a data lake, and to provide metadata and other contextual information to facilitate the analysis and search of datasets. Moreover, data discovery systems serve as a core component of other data-related tasks such as data integration, data cleaning, and machine learning. More concretely, given a query table or column, the objective is to find the most similar tables or columns in the data lake in a reasonable time. Throughout this report, the term dataset refers to a collection of tables, which are themselves collections of columns.

The field of data discovery research is concerned with improving one or more of the key components involved in the process, including data profiling, data catalog construction, and discovery operations. The following section explores relevant research that has focused on enhancing these components.



1) Explain in your own words what is a data lake and what would be the characteristics that convert a set of datasets into a data lke

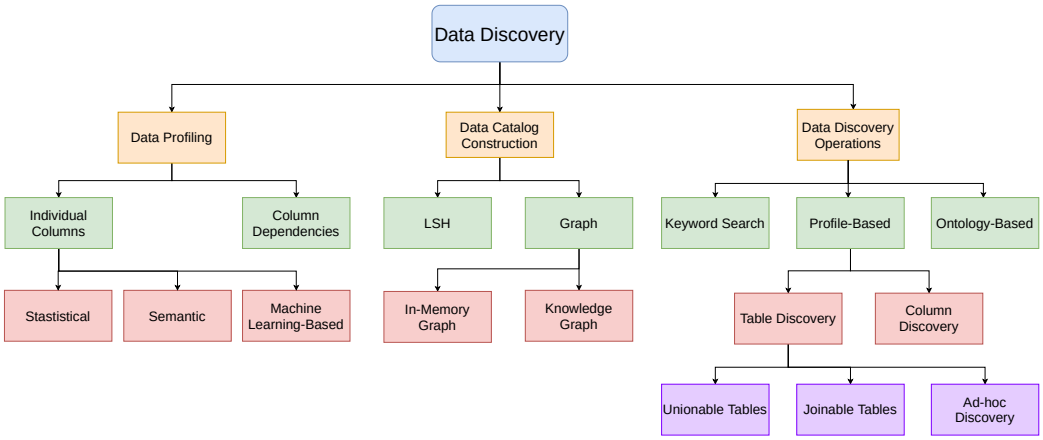



Figure 3: Taxonomy of research work on data discovery.

## 2.1 Taxonomy of Related Work

### 2.1.1 Data Discovery Components

There are three main components of data discovery systems for data lakes: data profiling, data catalog construction, and discovery operations.

**Data profiling** is the set of activities and processes designed to determine the metadata of a given dataset [4]. It involves the analysis of dataset properties including their names, schema, data types, raw values, value distributions, and semantics. The main goal of data profiling is to build a data profile (summary) per table or column, which is then utilized to perform different data science tasks such as data cleaning, data quality assessment, and data discovery. There are several data profiling techniques available on both the column and table levels. Section 2.1.2 describes these techniques in more detail.

**Data catalog construction** involves creating a searchable index over the datasets in the lake, making it easy for data scientists and analysts to discover, access, and use the data. Two popular approaches for constructing data catalogs in data lakes are LSH indexes and graphs. These approaches are described in more detail in section 2.1.3.

**Data discovery operations** or algorithms are the main techniques for discov-

ering and recommending relevant datasets within a data lake. These operations rely on both the generated data profiles and the constructed data catalog to effectively search and explore the data lake. The range of data discovery operations includes simple keyword searches as well as more complex ontology and profile-based searches. Sections 2.1.4, 2.1.5, and 2.1.6 explain these methods in more detail.

### 2.1.2 Data Profiling Techniques

Data profiling involves the analysis of datasets and the extraction of their metadata as data profiles. It is an essential step in data discovery to understand and summarize datasets in a data lake. In practice, data profiling is performed on two levels: individual columns and column dependencies [4]. Individual column analysis refers to building summary profiles about columns in the tables of a data lake in isolation from other columns. Popular techniques for individual column analysis include statistical, semantic, or machine learning-based analyses. Alternatively, column dependencies involve discovering relationships between columns in the same or different tables. Examples of such relationships include unique column combinations, functional dependencies, and inclusion dependencies.

**Statistical methods** are the simplest profiling techniques that are widely used in practice. Some statistical methods are valid for all data types (i.e. numerical and non-numerical) such as the number of rows (i.e. cardinality), number of missing (or null) values, and number of unique values. Other statistics are data-type-specific. For example, mean, median, quartiles, standard deviation, min, and max are typical summary statistics for numerical columns. Moreover, histograms and value distributions offer in-depth comprehension of numerical columns [25].

**Semantic analysis** is an essential profiling technique, especially for datasets with non-numerical columns. A basic form of semantic analysis is the use of regular expressions to identify specific value patterns [44]. For instance, columns containing dates, postal codes, or emails are straightforward to identify. Moreover, semantic analysis includes grouping textual columns into semantic domains. [51]

In [x], . . .

explain a methodology of grouping columns having the same meaning. Additionally, named entity recognition (NER) models [48] are becoming more popular in identifying semantic domains. Named entity recognition is the task of identifying entities in the world that appear in a text. Such entities include, for example, person, location, or organization names. Finally, a standard technique for understanding the meaning of textual columns, especially with long textual values, is the use of word embeddings such as Word2Vec [37] and GloVe [43]. Word embeddings offer a low-dimensional, dense vector representation of words based on the premise that words with similar meanings will have vectors with smaller distances between them; they have been shown effective in numerous natural language processing (NLP) tasks.

**Machine learning-based** techniques have become increasingly popular in various applications in recent years. Data profiling is one such area where machine learning has been used to perform tasks such as data type detection [24], data column dependency discovery [45], and data quality assessment [16], among others. A particularly important application of machine learning in data profiling is table representation learning, which involves using deep learning models to summarize tables into low-dimensional, dense vector representations, similar to word embeddings but for tables. This approach replaces the use of hand-crafted statistics for building column profile summaries, which may fail to summarize columns holistically in certain situations as indicated in [24]. An early work for table representation learning is the semi-supervised approach proposed by [38], which trains shallow neural networks to learn column embeddings. These embeddings have been shown to be of close distance for columns measuring the same variables, even with different distributions as shown in Figure 4. Other recent attempts at table representation learning include the use of language models such as pre-trained BERT models [17] and purposely designed transformer models [22]. Overall, table representation learning is an active area of research.

**Column dependencies** are the set of data profiling techniques aimed at identifying potential relationships between columns both within and across tables. An



- Data aggregation

1) How precise are these mechanisms for profiling? How can one identify bias or unfair profiling?

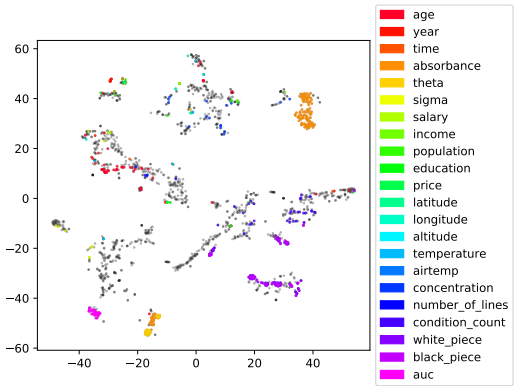



Figure 4: The approach of [38] generates column embeddings that are close in distance for columns that measure the same variables.

example of dependencies within a table is identifying a unique column combination, i.e. the set of columns that uniquely identify a table [47, 2]. Examples of dependencies across tables include functional dependencies [14] and inclusion dependencies [11]. Functional dependency between two columns $A$ and $B$ refers to the existence of a function such that all pairs of records with the same values in $A$ are mapped to the same values in $B$ [23, 3, 41]. Moreover, an inclusion dependency between two columns $A$ and $B$ exists of all values of $A$ also occur in $B$ [35, 28, 46].

### 2.1.3 Data Catalog Construction Methods

Many data discovery systems use data catalogs to improve search processes. While not mandatory, these catalogs enhance search efficiency by offering a searchable index covering the datasets in a data lake. This allows data scientists and analysts to easily find, access, and use the data. In data discovery systems, two common approaches for creating data catalogs include locality-sensitive hashing (LSH) indexes and graph-based structures.

**Locality-sensitive hashing (LSH)** is a method of indexing high-dimensional data points to enable efficient similarity-based searching. It is particularly useful when dealing with a large number of columns or tables, where traditional tech-

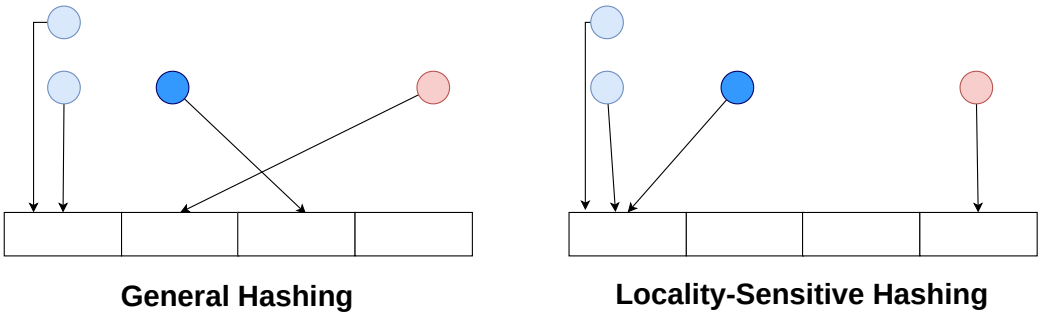



Figure 5: An illustration of the difference between general and locality-sensitive hashing functions. In general hashing, only identical items are mapped to the same bucket. In locality-sensitive hashing, both identical and similar items are mapped to the same buckets while maintaining a high distance from highly dissimilar items.

niques such as brute-force search do not scale due to the high computational cost. [30] The basic idea behind LSH is to partition the search space into disjoint "buckets" and map similar data points to the same bucket with high probability while mapping dissimilar data points to different buckets with high probability. This is achieved by constructing a family of hash functions that satisfy certain properties, such as locality sensitivity and pairwise independence. That is, unlike general hashing functions, where it is desired to decrease the probability of collision, hash functions used for LSH are designed to increase the probability of collision between similar items. Figure 5 illustrates the difference between general and locality-sensitive hashing.

For data discovery systems, LSH indexes are often used over the generated data profiles to efficiently detect similar tables or columns. Two popular LSH hash functions used in data discovery are MinHash [10], which estimates the Jaccard Similarity between two profiles, and random projections [12], which estimates the cosine similarity between two profiles. The latter is more suitable for numerical columns while the former is suitable for non-numerical or textual columns. LSH is widely used due to the significant improvement in similarity search. For N data profiles, LSH indexes could be used to determine all similar profiles in $O(N)$

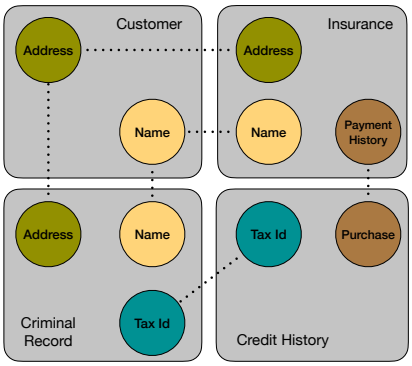



Figure 6: Example data catalog generated by SiMa [29] for four tables. Nodes correspond to columns, while edges represent detected similarities between them.

runtime. The brute-force approach, i.e. all-pair comparison has a runtime of $O(N^2)$. LSH indexes are used in several data discovery systems such as Aurum [20], $D^3L$[8], Table Union Search [39], and others.

**Graphs** are a type of data structure that is used to construct data catalogs in data discovery systems. In graphs, nodes represent entities and edges represent possible relationships between them. The definition of entities and relationships is a design choice based on the objectives and granularity of the system. Some of the common entities in data discovery systems are dataset metadata such as schema or columns. For instance, in Aurum [20], nodes represent columns, tables, and datasets, while edges represent hierarchical relationships and similarity relationships. In SiMa [29], nodes represent columns, and edges represent inferred similarities between them. An example of a data catalog generated by SiMa is shown in Figure 6. Graph-based data catalogs can be used to perform various types of queries, such as lineage tracing, impact analysis, and dependency analysis. Existing data discovery systems that implement graph data catalogs use in-memory graphs. While in-memory graphs provide low-latency queries, they have several limitations. First, they do not scale well with the size of the data lake because memory is normally limited in size. In addition, in-memory graphs do not provide a universal query language, which makes it more challenging for data

scientists to utilize the constructed catalog. One example of this is Aurum [20], which provides a custom query language named SRQL. Another approach to implementing graph catalogs is the use of knowledge graphs [21], which provide a universal query language (SPARQL) and scale well to billions of nodes and edges. There is yet a data discovery system that systematically uses knowledge graphs as a data catalog.

### 2.1.4 Data Discovery Operations

Data discovery operations are the main techniques, where the generated profiles and data catalog are utilized to discover relevant datasets in a data lake. Discovery algorithms differ in complexity based on the objective of the system. Three common approaches are keyword search, ontology-based discovery, and profile-based discovery.

**Keyword search** is the most basic form of data discovery and is widely implemented in various systems such as Google Dataset Search [40][1], Amundsen [2], Open Data Discovery [3], among others. It allows data scientists and analysts to explore a data lake in a natural way using keywords or key phrases. The data discovery system matches the supplied keywords and phrases against the metadata of datasets, tables, or columns in the data lake. Keyword search is suitable for settings where the data lake is of relatively small size and the data is well-structured and documented. In a practical setting, however, where a data lake contains thousands of datasets of potentially low quality (e.g. missing attribute names), keyword search becomes inefficient due to the large number of results returned. Therefore, advanced data discovery algorithms are developed to reduce the number of false positives returned by the system.

**Ontology-based discovery** involves the use of ontologies to model the domain of interest and to define the relationships between the concepts and enti-

[1] https://datasetsearch.research.google.com/
[2] https://www.amundsen.io/
[3] https://opendatadiscovery.org/



1) What is the definition of ontology.

ties in the data. Ontologies are knowledge graphs that define a set of concepts, their relationships, and the rules that govern them. Data discovery systems utilize open ontologies like DBPedia [7] and YAGO [42] or enterprise ontologies like the Google Knowledge Graph [4] and the Amazon Product Knowledge Graph [5]. In ontology-based data discovery systems, datasets are indexed and annotated using the concepts and relationships defined in the ontology. This allows users to search for datasets based on their relevance to a particular concept or topic, thus reducing the number of false positives compared to keyword search. Ontology-based discovery is used in state-of-the-art data discovery systems. For instance, SANTOS [27] uses both an open ontology and a synthesized ontology for the data lake to facilitate the discovery of relevant tables based on the common relationships between the columns of a query table. The use of ontologies also helps to address the issue of data heterogeneity, where datasets may use different terminology to describe similar concepts. By defining a common set of concepts and relationships, ontologies can help to standardize the terminology used to describe datasets, making it easier to discover and compare them.

**Profile-based discovery** uses column or table profiles to discover relevant data items. While ontology-based discovery relies on metadata (i.e. schema) of the datasets in the lake, profile-based discovery uses both metadata and *content* of the query table to discover relevant datasets. This is achieved by defining a similarity threshold and comparing the generated profiles. If two profiles have a similarity score above the threshold, the corresponding data items are similar. A higher similarity score indicates the items are more similar. In the same way, if two tables have more similar columns, they are likely to be more similar. As discussed in Section 2.1.3, two possible methods of comparing column profiles are pairwise comparison and approximate nearest neighbors via LSH indexes. The former is an exhaustive search with quadratic time complexity. In practice, LSH indexes are used with an appropriate hash function due to the linear time complexity. Not

[4] https://blog.google/products/search/introducing-knowledge-graph-things-not/
[5] https://www.amazon.science/blog/building-product-graphs-automatically



1) Why did you mention in the previous section the lack of data discovery systems using knowledge graphs but here there are several ontology-based systems, which are using knowledge graphs.

unlike ontology-based discovery, several state-of-the-art data discovery systems utilize profile-based discovery. Examples include Starmie [19], a profile-based discovery system that utilizes pre-trained language models to compute column profiles. In general, profile-based discovery is more common in data discovery systems than ontology-based discovery.

### 2.1.5 Granularity of Profile-Based Discovery

Profile-based data discovery systems differ in granularity based on the design and discovery objectives of the system. The most common two levels of discovery are table discovery and column discovery.

**Table discovery** is a high granularity for data discovery systems. In table discovery, the data scientist provides a query table and the data discovery system recommends similar tables and provides the similarity score to the query table for each result table. The data discovery system does not provide more explanation behind the similarity score or the specific columns that are matching in the tables. Examples of table discovery systems include TUS [39], $D^3L$ [8], and SANTOS [27].

**Column discovery** systems provide more information by operating on the low granularity of columns. The data scientist provides a query column or table and the data discovery system recommends similar tables or columns with their similarity scores based on matches on the column level. Thus, a column discovery system helps the data scientist understand why specific tables are more similar than others. In addition, column discovery systems are more practical for scenarios where tables have a large number of columns. Examples of column discovery systems include Aurum [20] and Starmie [19].

### 2.1.6 Table Discovery Objectives

The discovery objective in a data discovery system is a design choice that determines the algorithm and techniques by which relevant tables are retrieved from the data lake. The most common discovery objectives are table unionability and

joinability. In addition, ad-hoc table discovery is supported by some discovery systems. While these are the most common discovery objectives, some systems like $D^3L$ [8] use the more general definition of table relatedness, which expresses that two tables are related without specifying the type of similarity relationship between them.

**Unionable table discovery** is concerned with discovering tables that can be unioned with the query table. According to [39], two tables are unionable if they share columns that have values from the same domain so that they can be concatenated row-wise. Discovering unionable tables is essential for several data science tasks as data scarcity is an often-cited issue. Unionable tables allow a data scientist to enrich their table with more data instances, i.e. rows. For example, consider a scenario where the goal of a data scientist is to predict house prices in a certain city given open government data that describe the house features. In this case, a unionable table could be the house prices from previous years or the house prices for another city published by the same government sector. By unioning these tables, the data scientist can train a more accurate model to predict house prices. Examples of unionable table discovery systems include TUS [39], SANTOS [27], and Starmie [19].

**Joinable table discovery** is concerned with discovering tables that can be joined with the query table. Joinable tables can be concatenated column-wise and according to [53], two tables are joinable if they have at least one column with high overlap, i.e. the columns have many values in common. Similar to unionable tables, discovering joinable tables is vital in several data science tasks. Unionable tables allow a data scientist to enrich their table with more features, i.e. columns. Consider the previous example of predicting house prices given open government data. The data scientist can use another table for crime rates, which allows them to have an extra feature of the crime rate in the house neighborhood. As house prices are directly correlated with this feature, this allows the data scientist to have a more accurate model. Examples of joinable table discovery systems include Auto-Join [52] and JOSIE [53].



1) Explain graphically the concepts of unionable tables and joinable tables

**Ad-hoc discovery** is useful in scenarios where the data scientist aims to find relevant tables beyond joinable and unionable. Example scenarios include exploring tables using both keyword search and profile-based discovery or exploring tables with a certain number of matching columns to the query table. Data discovery systems with one objective such as unionability or joinability are inflexible and cannot support such operations. However, these operations can be supported using appropriate data catalogs like graph catalogs. For example, in Aurum [20], the data scientist can supply a custom query that meets their discovery requirements. One limitation of Aurum, however, is that the data scientist needs to learn a custom query language (SRQL), which reduces their efficiency. Knowledge graph catalogs provide a universal query language that supports ad-hoc discovery operations (SPARQL). There is yet a data discovery system that utilizes knowledge graphs to perform ad-hoc discovery.

## 2.2 Examples of Recent Work

This section describes the data discovery techniques used in four exemplar systems.

### 2.2.1 $D^3L$

$D^3L$ [8], a short-hand for Dataset Discovery in Data Lakes, is a standard data discovery system in terms of techniques and objectives. Given a target table and a data lake, $D^3L$ returns the top-$k$ similar (both unionable and joinable) tables in the data lake, where $k$ is a parameter.

To measure the similarity between two tables, $D^3L$ first measures the similarity between their columns using five types of hand-crafted features. Then, the column similarities are aggregated to have a similarity score for each pair of tables. The collected features differ based on the data type of a column. For both textual and numerical columns, the following features are collected:

- n-grams of Column Name $\mathbb{N}$: Given a column name, construct the set of

n-grams (the authors use n=4). For example, for a column named `target`, the output is `{targ, arge, rget}`.

- Format $\mathbb{F}$: Given the column values and a set of predefined regular expressions, construct the set of expressions that match one or more values in the columns. Note that numerical values match the expression `[0-9]+`. For an alphanumeric column, the constructed set could be `{[A-Za-z0-9]+, [A-Z]+, [a-z]+}`.

For textual columns, the following features are collected:

- Informative Tokens $\mathbb{V}$: Given the set of all tokens existing in the column values, construct the set of the most informative tokens, i.e. tokens that uniquely exist in this column and not other columns (similar to the concept of TF/IDF). For instance, if a column has the values `{'Concordia University', 'Mount Royal Park', 'Old Port of Montreal'}`, the informative tokens could be `{'Concordia', 'Mount', 'Royal', 'Port', 'Montreal'}`.

- Word Embeddings $\mathbb{E}$: Construct the word embedding vector of a textual column by calculating the word vector using a word embedding model (the authors use FastText [9]) for each value and aggregating over all values of a column.

For numerical columns, the following feature is collected:

- Distribution Distance $\mathbb{D}$: To capture the domain of values in a numerical column, use the Kolmogorov-Smirnov statistic (KS)[15]. The KS statistic is a measure of the distance between two numerical distributions.

To measure the similarity between two columns, the Jaccard similarity is used for $\mathbb{N}, \mathbb{F}$, and $\mathbb{V}$, the cosine similarity is used for $\mathbb{E}$, and the KS statistic is used for $\mathbb{D}$. These distances are concatenated into a distance vector: $\{D_{\mathbb{N}}, D_{\mathbb{F}}, D_{\mathbb{V}}, D_{\mathbb{E}}, D_{\mathbb{D}}\}$.

The similarity between two tables is calculated as the distance between their distance vectors, which is a value $\in [0, 1]$. For a target table $\mathcal{T}$, the top-$k$ similar tables are the ones that have the lowest $k$ distances to $\mathcal{T}$.

### 2.2.2 Aurum

Aurum [20] is a system for the exploration of data lakes. Similar to $D^3L$ [8], Aurum supports the discovery of unionable and joinable tables to a given query table (named relationship constraints). In addition, Aurum supports discovering tables via keyword search or the number of unique values in a column (named property constraints).

Aurum's discovery operations revolve around a central navigational data structure called the Enterprise Knowledge Graph (EKG). In this graph, columns, tables, and data sources are nodes, while edges represent either similarity relationships between columns like joinability and unionability, or hierarchical column-table and table-source relationships.

Building the EKG requires determining the similarity relationships between all pairs of columns. Aurum follows a two-stage process with the granularity of columns to avoid the costly all-pair comparison. In the first stage, column information, including data type, value distribution, value signature (MinHash), and column name signature (TF-IDF), is collected. In the second stage, Aurum detects the similarities between columns using approximate nearest neighbor search based on their sketches. To achieve that, Locality Sensitive Hashing (LSH) is used with Jaccard similarity for MinHash signatures or cosine similarity for TF-IDF signatures. In addition, the similarity scores between column signatures are used to annotate the corresponding similarity edges. That is, columns that are more similar have higher similarity scores on their edges. The end result of this two-stage process is the EKG, built in $O(n)$ time by having multiple swaps over the data.

To perform data discovery operations, the authors propose to query the EKG via a purposely designed graph query language named the Source Retrieval Query

```
contentSim(table: str) =
    cols = columns(table)
    return jaccardContentSim(cols)
           OR cosineSim(cols)
           OR rangeSim(cols)
results = contentSim("HousingPrices")
```

Figure 7: A query in SRQL developed by Aurum [20]. `jaccardContentSim`, `consineSim`, and `rangeSim` are different column similarity measures. This query discovers columns that have content similarity with the query table "HousingPrices".

Language (SRQL). SQRL supports discovering tables and columns based on keyword search, column similarities, or a shared path in the graph. An example SRQL query is shown in Figure 7

### 2.2.3 SANTOS

SANTOS [27] is a recent semantics-based approach for the problem of table union search, i.e. discovering unionable tables in large data lakes. Unlike prior approaches to unionable table discovery like [39], SANTOS defines table unionability in terms of both raw values and table semantics. SANTOS uses two types of semantics to discover unionable tables: column semantics and relationship semantics. Column semantics refers to the conceptual domain to which the values in a column belong. A column might have one or more semantic labels/annotations. For instance, consider table (a) in Figure 8; the column "City" might have the semantic labels: `city`, `location`, or `address`. Column semantics are determined based on both the column name and values using open ontologies such as DBPedia [7] and YAGO [42]. Column semantic labels have been used in prior table union search approaches.

In addition to column semantics, SANTOS defines unionability in terms of relationship semantics, which refer to the logical association or connection between columns in the same table. For example, for table (a) in Figure 8, the column "Park

Name" has the semantic relationships `isSupervisedBy` and `isLocatedIn` with the columns "Supervisor Name" and "City", respectively. If table (a) is the query table and tables (b),(c) reside in the data lake, using column semantics only results in table (c) being the most similar even though (c) is about celebrities and (a) is about parks. However, since table (b) shares more relationship semantics with (a) than table (c) does, using relationship semantics achieves more accurate results in such scenarios.

SANTOS determines relationship semantics between columns by matching against an open ontology (YAGO [42]). For each table in the data lake, SANTOS constructs a *semantic graph*: a graph whose nodes are columns and edges are the semantic relationships between them. When given a query table, SANTOS creates its semantic graph then performs sub-graph matching over the data lake. SANTOS calculates a similarity score for all tables in the lake, where the scoring function depends on the number of column and relationship semantics in common. SANTOS recommends the top-k matching tables in the lake.

For some data lakes, the column semantics cannot be found in YAGO. To tackle this issue, SANTOS constructs a synthesized ontology for the tables in the data lake and uses it along with the open ontology to improve the table discovery. SANTOS demonstrates that using both open and synthesized ontologies achieves state-of-the-art results in table union search.

### 2.2.4 Starmie

Starmie [19] is a recent approach for unionable table discovery. Unlike other approaches for table union search, Starmie uses machine learning-based data profiling. Starmie utilizes language models like BERT [18] to learn dense, high-dimensional vectors (embeddings) of columns. Columns that have similar values, and are, therefore, unionable, have similar embeddings. Language models like BERT have proven recently to be successful in various NLP tasks. Language models are firstly pre-trained on a large corpus of natural language and then fine-tuned on downstream tasks, which requires high-quality data.



1) What is achieving state-of-the-art? What are the metrics that help establishing the SOTA performance in these evaluations?

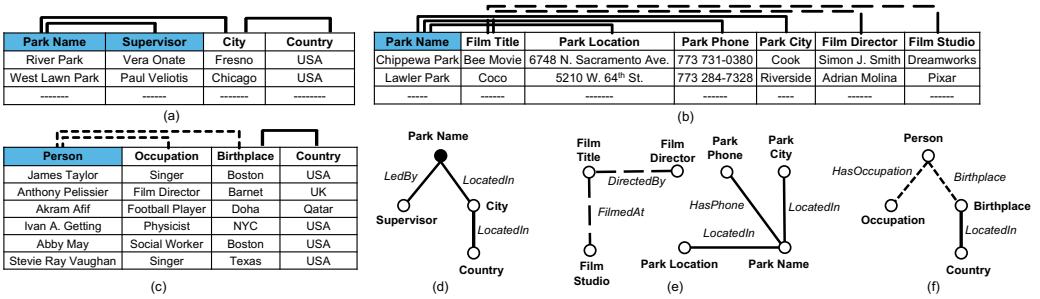
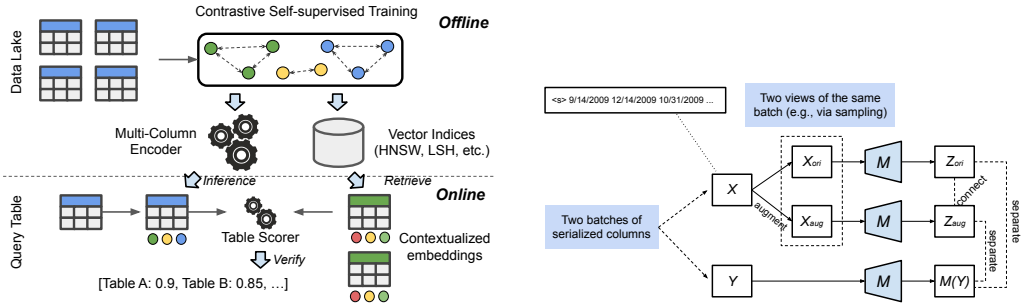



Figure 8: An example data lake demonstrating the relationship semantic-based approach of SANTOS [27]. Table (a) is about parks, table (b) contains information about parks and films shown in the park, and table (c) is about celebrities. The relationship semantic graphs of the three tables are shown in (d), (e), and (f), respectively.

Table union search in Starmie is divided into two stages: offline and online (shown in Figure 9.a. In the offline stage, a language model is trained to learn column embeddings for all columns in the data lake. Instead of embedding columns of a table in isolation from each other, Starmie utilizes a multi-column table encoder to generate contextualized column embeddings. That is, the embedding of a column depends on both its values and `context`, i.e. other columns in the same table. This leads to a more accurate matching. In addition, Starmie uses contrastive learning [13], a recent self-supervised learning approach to train ML models for data representation. As depicted in Figure 9.b, models trained via contrastive learning take triples as input, where the first two items (columns in the case of Starmie) are similar to each other but dissimilar to the third item. The model learns to generate similar representations for the first two items and a far representation for the third. Contrastive learning requires high-quality training data. Starmie uses data augmentation methods to generate positive samples. Example augmentation methods include sub-sampling or shuffling a column. Negative samples are obtained using a uniform random sampler over all columns in the data lake.

The second stage is the online stage in which a query table is given and union-




(a) (b)

Figure 9: Starmie [19] utilizes a two-stage approach (a) for table union search and uses contrastive learning (b) to fine-tune a pre-trained language model for column embeddings.

Table 1: Statistics of `SmallerReal` and `Synthetic` datasets used for evaluation by [8].

| Statistic | `Smaller Real` | `Synthetic` |
|---|---|---|
| Size (GB) | 1.3 | 4.8 |
| Total No. tables | 654 | 5,044 |
| Total No. columns | 8,767 | 54,758 |
| Avg. No. related tables per table | 110 | 260 |
| Avg. No. rows per table | 16,926.4 ($\pm$ 102,581.8) | 1,914.9 ($\pm$ 1837.6) |

able tables in the data lake are to be found. Starmie computes the contextualized embeddings for the columns in the query table and performs a nearest neighbor search over columns in the lake using Hierarchical Navigable Small World (HNSW). HNSW [33] is a graph-based method recently proposed for approximate nearest neighbor search as an alternative to hash-based techniques like LSH. HNSW greatly improves the query time of Starmie with a minimal loss in accuracy. On a benchmark of table union search, Starmie is shown to outperform the state-of-the-art systems including SANTOS [27] and $D^3L$ [8].

## 2.3 Data Discovery Evaluation

Data discovery systems are mainly evaluated based on the effectiveness of discovery, i.e. the accuracy of discovered tables in a data lake. Moreover, they are evaluated based on their efficiency, i.e. the time required to process and recommend data items.

To evaluate data discovery systems, benchmark data lakes have been developed such as the ones proposed by [8] or [27]. Benchmark data lakes are either real-world or synthetic. Real-world benchmarks are sampled from open data portals such as OpenML [49] or government data portals like the US [6] or Canada [7] data portals. Real-world benchmarks closely resemble data encountered in practice with diverse data types, domains, and sizes. Additionally, real-world benchmarks are not necessarily preprocessed or cleaned; their tables potentially contain missing values or inaccurate column names. Therefore, using real-world benchmarks gives a more accurate evaluation of data discovery systems in practical scenarios. However, real-world benchmarks require manual annotation: for each table in the data lake, all related tables have to be manually labeled, which is time-consuming. Furthermore, for column discovery systems, pairs of related columns also have to be annotated manually. This limits the use of real-world benchmarks to either relatively small data lakes, or for evaluating the efficiency, which does not require labelling. To overcome this limitation, synthetic benchmarks are developed by collecting a large number of tables and automatically performing random projections such as sampling rows or columns and splitting a table into multiple tables. Synthetic datasets do not require manual annotation and are larger in size but are not as practical as real-world datasets. In practice, both real-world and synthetic benchmarks are used to evaluate data discovery systems. Table 1 contains a comparison between a real-world and a synthetic dataset used by $D^3L$ [8]. As shown, the real-world dataset is more variable in the number of rows but the synthetic dataset is larger in size and has more related tables on average, allowing

[6] https://www.data.gov
[7] https://open.canada.ca

to evaluate top-k table discovery for larger values of $k$.

The most common evaluation metrics for data discovery systems are precision, recall, F1-Score, and Mean Average Precision (MAP). All these metrics depend on the notions of true positives, false positives, and false negatives. Given a query table $q$, a data lake $\mathcal{D}$, and a set of tables $\mathcal{T} \in \mathcal{D}$ recommended by the data discovery system, true positives refer to tables $t \in \mathcal{T}$, that are related to $q$. Similarly, false positives refer to tables $t \in \mathcal{T}$, that are not related to $q$ but are falsely retrieved by the system. Finally, false negatives refer to tables $t \in \mathcal{D}$, that are related to $q$ but are not retrieved by the system, i.e. $t \notin \mathcal{T}$.

**Precision** is the ratio of relevant tables retrieved out of all retrieved tables:

$$precision = \frac{TP}{TP + FP} \tag{1}$$

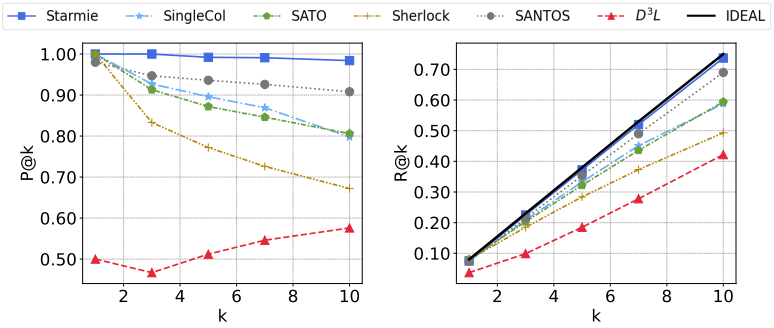
Systems with high precision retrieve mostly relevant tables. **Recall** is the ratio of relevant tables retrieved out of all relevant tables in the data lake:

$$recall = \frac{TP}{TP + FN} \tag{2}$$

Systems with high recall retrieve most of the relevant tables in the data lake. **F1-Score** is a single metric that combines both precision and recall, balancing the trade-off between the two metrics:

$$F1 = \frac{2 \times precision \times recall}{precision + recall} \tag{3}$$

It is common to evaluate the previous metrics at different values of $k$, where $k$ is the number of related tables desired. As a result, precision@k, recall@k, and F1@k are also popular metrics. Moreover, **Mean Average Precision (MAP)** is




**(a) $P@k$ on SANTOS Small** **(b) $R@k$ on SANTOS Small**

Figure 10: A comparison between Starmie [19] and the state-of-the-art data discovery systems in terms of precision and recall.

used to summarize the precision at the different values of $k$ in a single number:

$$MAP = \frac{1}{k_{max}} \sum_{k=1}^{k_{max}} precision@k \tag{4}$$

The example systems mentioned in Section 2.2 have been compared against each other on different benchmarks. Figure 10 illustrates a comparison by [19] showing Starmie against the state-of-the-art data discovery system on a small real-world benchmark. Starmie is shown to outperform other systems including SANTOS and $D^3L$. Moreover, in a comparison by [8], $D^3L$ is shown to outperform Aurum.

Table 2: A Scalability comparison by SANTOS [27] showing the indexing and query times compared to $D^3L$.

| Benchmark | Method | Indexing | Query (sec) |
|---|---|---|---|
| **SANTOS LARGE** | $D^3L$ | 7 hr 7 min | 177 (13.0 – 325.0) |
| | $\text{SANTOS}_{\text{Full}}$ | 21 hr 59 min | 35.8 (0.21 – 57.2) |

Another dimension of evaluation is whether the data discovery techniques can scale to large data lakes. Table 2 shows a scalability comparison between SANTOS and $D^3L$ on a data lake of size 11,090 tables. Indexing time is the time

required to analyze and profile the tables in the data lake and construct data catalogs. Query time is the time required to retrieve related tables in the data lake given a query table. SANTOS trades off more indexing time with less query time.

# 3 Semantic Code Understanding

One of the main elements of data science platforms besides datasets is the code written by data scientists to perform various tasks. Data scientists develop data science pipelines, short scripts of typically a few hundreds of lines of code [8], which perform specific data science tasks. A data science pipeline contains one or more of the following tasks: data analysis, data preprocessing and preparation, data modeling i.e. machine learning, and model evaluation. Data scientists share their created pipelines either on open or enterprise platforms and collaborate e.g. to enhance model utility on a specific dataset. This led to the explosion of openly available pipelines written for different datasets, tasks, and business domains. For instance, Kaggle, the most popular data science platform, contains hundreds of thousands of pipelines written by both experts and beginners alike. Therefore, this open knowledge, if properly analyzed and represented, has the potential to enable novel applications and methods that help data scientists utilize the pipelines to the fullest. Examples include making the pipelines easier to explore by highlighting the different tasks performed in each pipeline or assisting novice data scientists in choosing the right techniques based on the knowledge of their experienced colleagues. Unfortunately, existing portals and platforms have no or little support to assist data scientists in easily exploring and systematically discovering data science pipelines. Instead, they offer only isolated and limited support for pipeline recommendation based on basic filters such as the creation date or the number of votes.

One of the primary challenges in providing such support is to understand the semantics of the scripts and represent them in a machine-readable format. For-

[8] Based on a sample of 15,000 pipelines from Kaggle.

tunately, numerous techniques have been developed for general-purpose code understanding, which enabled several applications such as code search, code automation, refactoring, bug detection, and code optimization [5]. In this chapter, the area of semantic code understanding is explored with a focus on the techniques and methods that are applicable to data science pipelines.

Semantic code understanding involves two main tasks: code analysis and semantic representation. **Code analysis** refers to the set of techniques aimed at analyzing source code to collect more information about its logical and semantic structures. **Semantic representation** involves storing the captured information from code analysis in a suitable data structure that allows for retrieval and reuse of information.

## 3.1 Taxonomy of Related Work

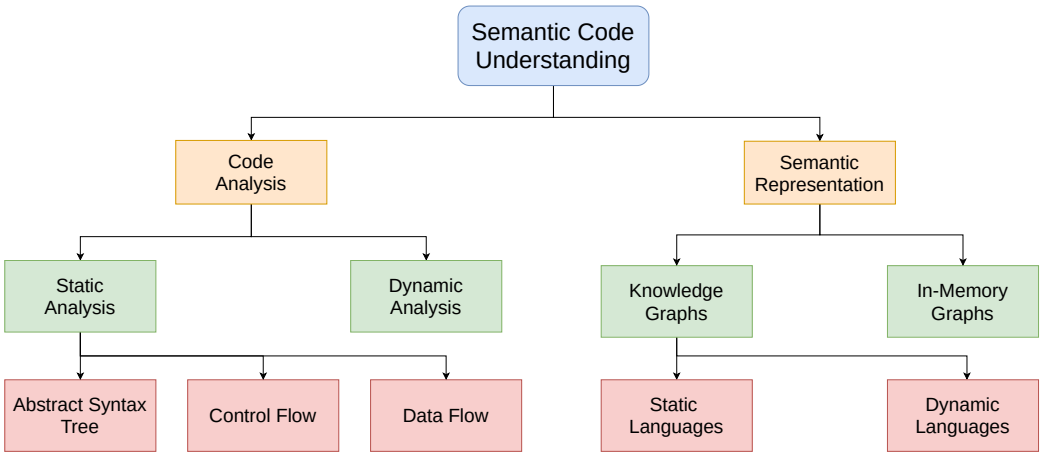



Figure 11: Taxonomy of research work on semantic code understanding.

### 3.1.1 Code Analysis Approaches

When analyzing data science pipelines, there are two possible approaches: dynamic and static code analysis. **Dynamic code analysis** means studying a pro-

gram's behavior while running. It involves running the program with different inputs and collecting information at each call such as CPU usage, memory traces, dataset statistics, variable types, etc. Dynamic code analysis provides a thorough understanding of the behavior of the program at each code statement. That said, dynamic code analysis has two major limitations. Firstly, it is input-specific; in order to cover all code statements, various inputs are required to be tested out. Second, running programs or scripts can be time-consuming, especially for data science pipelines; reading datasets, computing features, and building machine learning models are all time-consuming tasks.

The second code analysis approach is **static code analysis**, which refers to analyzing source code without running it, i.e. based only on the syntax. Unlike dynamic analysis, static code analysis is time- and memory-efficient, since it does not involve code execution. Moreover, it is independent of a program's input. However, static analysis has one major drawback; the collected information is far less than that collected by dynamic analysis. For the problem of semantic code representation of data science pipelines, static analysis can scale well to the hundreds of thousands of publicly available pipelines but needs to be enriched with techniques that are tailored for the libraries used in data science pipelines.

### 3.1.2 Static Code Analysis Techniques

Several techniques are used for understanding data science pipelines via static code analysis. First, **Abstract Syntax Tree (AST)** analysis involves the study of a program's syntax and structure. An abstract syntax tree is a tree-like representation of a program code that captures the syntactic structure and relationships between various elements of the code including classes, functions, variables, and function parameters, among others. AST analysis is useful in understanding the general structure of a data science pipeline.

**Control flow** analysis is the study of the logical flow between program expressions. The outcome of control flow analysis is a Control Flow Graph (CFG), which is a directed acyclic graph (DAG) representation, where nodes are program

statements and edges are the logical flow (order) between them. An edge exists from statement A to statement B if A is directly followed by B. For data science pipelines, control flow analysis is essential in understanding the order of operations like data preparation and data modeling.

**Data flow** analysis is the study of how a program data (variables) is utilized and manipulated throughout the program. It involves constructing a data flow graph, which is a directed acyclic graph (DAG), where nodes are program statements and edges are the data flow between them. An edge exists from statement A to B if B uses a variable that is created or used in A. For data science pipelines, data flow analysis is critical since it highlights how a dataset is used and manipulated throughout the pipeline.

### 3.1.3 Code Representation Methods

The outputs of static code analysis techniques are the abstract syntax tree (AST), control flow graph (CFG), and data flow graph. A natural representation that encapsulates these outputs is a single directed acyclic graph (DAG) whose nodes are statements, functions, classes, and packages, while edges are structural relationships, control flow, and data flow. Similar to graph data catalogs described in Section 2.1.3, two approaches exist for representing and handling graph code representations. **In-memory graphs** provide low-latency queries but have limited usability as they don't have a universal query language or storing mechanism. In contrast, **knowledge graphs** [21] support a universal query language (SPARQL) and scale well to graphs of billion nodes and edges. In addition, they support integrating all data science pipelines on a platform in a single unified graph, while keeping each pipeline graph separate via the use of named graphs. Therefore, knowledge graphs are a more suitable representation of pipelines in data science platforms.

### 3.1.4 Code Representation as Knowledge Graphs

Several works have been proposed for the problem of knowledge graph construction from source code [31, 1, 50, 6, 32]. All of these approaches use static code analysis to understand source code. Moreover, they can be classified into two categories based on whether the target programming language is a statically-typed language or a dynamically-typed language. **Statically-typed languages** such as Java, C++, and C# require variable and function return types to be declared at compile time. As a result, using static code analysis collects detailed information about the source code. In contrast, **dynamically-typed languages** such as Python, Ruby, and JavaScript do not require type declaration and infer the variable and function return types at compile time. This makes static code analysis challenging as little information can be inferred about the source code.

Data science pipelines are mostly written in Python. Based on the MetaKaggle dataset[36], more than 91% of the pipelines on Kaggle are written in Python. This poses a challenge as the static analysis of Python scripts alone is not sufficient to construct rich knowledge graphs. Therefore, supportive approaches are required to accurately represent data science pipelines.

## 3.2 Examples of Recent Work

In this section, we describe recent approaches that utilize static code analysis for constructing knowledge graphs from source code.

### 3.2.1 CodeOntology

CodeOntology [6] is one of the early works for constructing knowledge graphs for source code. The main contribution of this work is the design of an ontology, i.e. graph schema, that captures the different semantics of source code. A well-designed ontology increases the expressivity of a knowledge graph and its integrability with open knowledge graphs like DBPedia [7].

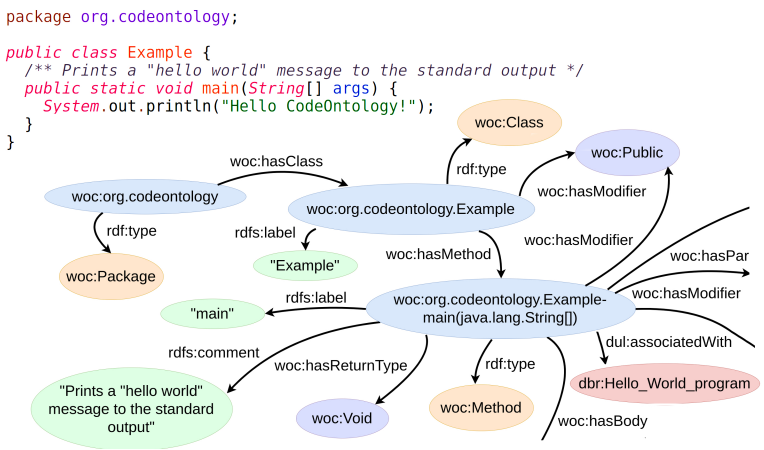


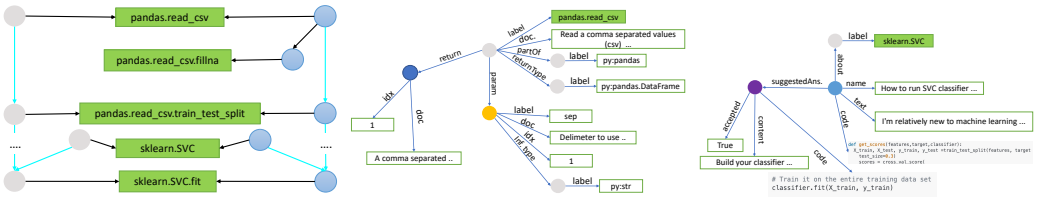

Figure 12: Sample "HelloWorld" program and its corresponding knowledge graph representation generated by CodeOntology [6].

The CodeOntology parser is designed for statically-typed programming languages like Java and uses abstract syntax trees (AST) for static code analysis. In this graph, nodes represent structural entities in source code such as projects, libraries, classes, methods, variables, statements, and expressions. Edges indicate structural relationships, node types, and node labels. An example "HelloWorld" Java program and its knowledge graph representation are shown in Figure 12. The ontology design proposed by [6] has laid the foundation for other systems like GraphGen4Code described in the next section.

### 3.2.2 GraphGen4Code

GraphGen4Code [1] is a toolkit designed for the construction of knowledge graphs from source code as well as related artifacts such as docstrings and forum discussion. GraphGen4Code is demonstrated to scale by constructing a knowledge graph containing 1.3 million Python programs and 47 million online forum posts.

The knowledge graph generated by GraphGen4Code consists of three parts. First, a code graph models the source code of input scripts. In this graph, there are two types of nodes: program expressions and external library functions. Edges




(a) (b) (c)

Figure 13: An example from the knowledge graph proposed by [1], which consists of (a) a code graph, (b) a docstring graph, and (c) a StackOverflow graph.

represent the control and data flows between program expression nodes or the invocation of external methods by program expressions (labels). Program graphs are stored separately to achieve isolation between input scripts. However, expressions in separate programs that call the same library functions are indirectly linked through the library function nodes. This increases the semantics captured in the graph. Second, a docstring graph models the structure of external libraries invoked in input scripts. In this graph, nodes represent packages, classes, functions, and parameter names of external libraries like `pandas` and `Scikit-Learn`. Edges indicate the logical structure between the elements, the default parameter values, and the return types of the functions. Third, an online forum discussion graph models the questions and answers about external libraries on online forums such as StackOverflow. In this graph, nodes represent library functions, questions, answers, and votes while edges indicate the connections between these elements.

The three graphs are not isolated from each other. Rather, they share nodes that represent library functions. That is, for an expression in a Python program, one can retrieve the default parameter values of library functions in the expressions as well as related StackOverFlow questions about these functions. Figure 13 shows examples of the three parts of the knowledge graph.

GraphGen4Code analyzes Python programs using WALA [9], a static analysis tool designed for multiple programming languages including Python, JavaScript,

[9] https://wala.github.io/

and Java. WALA implements general-purpose static analysis methods including abstract syntax tree analysis, control flow analysis, and data flow analysis. The code graph of GraphGen4Code has edges that indicate logical flow, data flow, dataset reads, parameter names, and parameter orders of all expressions in a Python program. GraphGen4Code attempts to alleviate the scarcity of information provided by static code analysis for Python by using the docstring graph to infer default parameter names and values.

Although GraphGen4Code is demonstrated on a large collection of Python programs scraped from GitHub, its semantic code modeling is lacking for data science pipelines. The general-purpose techniques used by GraphGen4Code are not tailored for data science pipelines, which have a specific structure and flow. As a result, GraphGen4Code consumes a significant amount of time and generates redundant, local information such as line and column numbers for statements and the order of parameters in a function. We ran GraphGen4Code on a collection of 14,000 pipelines from Kaggle. GraphGen4Code took 2,255 minutes to execute and generated a code graph of 97.5 million triples (edges).

# 4 Open Research Challenges

Despite the significant progress that has been achieved in the fields of dataset discovery in data lakes and semantic code abstraction, there is still room for further development in these techniques and their integration. In this section, some of the limitations of prior work and research opportunities are highlighted.

## 4.1 Dataset Discovery via Column Embeddings

Profile-based dataset discovery has been traditionally based on hand-crafted features including basic statistical measures and semantic features like word embeddings. The use of deep learning models to generate dataset profiles has seen limited attention. Deep neural networks have the potential to learn the most important

features of datasets without manual annotation. While some recent systems use pre-trained language models to embed columns [19], more work is required to compare them against purposely-designed deep neural networks. In addition, the usability of these embeddings in different settings e.g. for data types, could be studied.

## 4.2 Scaling Data Discovery

Data lakes, especially in enterprise settings, could reach hundreds of thousands of datasets. This presents a challenge for scaling data discovery techniques. Specifically, trade-off better query times for significantly greater indexing times. An open research question is whether both indexing and query times can be scaled simultaneously and whether representing the discovered relationships in suitable data structures like knowledge graphs can scale query time.

## 4.3 Semantic Abstraction of Data Science Pipelines

Prior research on code abstraction focuses either on static languages like Java or utilize general-purpose code analysis techniques. However, the majority of data science pipelines are written in Python, a dynamic language with limited information gathered from generic static code analysis. In addition, data science pipelines have a common program structure such as data analysis, data preparation, data modelling, etc. Moreover, the majority of data science pipelines use a limited set of data science libraries and frameworks. These unique properties could be exploited to develop code analysis techniques that abstract the most important aspects of data science pipelines such as the datasets or columns being read, data engineering operations performed, ML models built, and so on.

## 4.4 Linking Data Lakes and Data Science Pipelines

One of the most promising research opportunities is to explore whether the semantics of datasets and data science pipelines could be linked in a universal representation. The primary items in data scince platforms are datasets, consisting of tables and columns, as well as pipeline notebooks containing the operations performed on theses datasets. Linking the semantics of data science items has the potential to bring about novel applications for data science platforms. Example applications could include the automatic generation of ML pipelines, automatic prediction of suitable feature engineering techniques, semantic code clone detection for data science notebooks, among others. It remains an open question to identify suitable techniques for the extraction and representations of these semantics.

# 5 Conclusion

In this report, a literature survey has been conducted on research work related to two areas: data discovery and semantic code understanding. Data discovery is the problem of finding relevant datasets among a large collection stored in data lakes. Three major data discovery components have been detailed, namely, data profiling, data catalog construction, and discovery operations. Several data discovery systems that employ a variety of techniques and discovery objectives have been detailed including systems with state-of-the-art results on discovery benchmarks. The report also discussed the problem of semantic code understanding, i.e. representing the semantics of data science pipelines as graphs to automate various tasks. The general code analysis techniques have been explained including different static code analysis techniques. Moreover, related systems that abstract source code as knowledge graphs have been detailed.

It has been highlighted that existing techniques have several limitations and study either datasets or data science pipelines in isolation from each other. This motivates our future work to improve existing techniques by utilizing knowledge graphs. More importantly, data science artifacts need to be linked in a holistic

representation to facilitate their exploration and the automation of various tasks on top of them. Our future research will focus on building a linked data science platform to overcome these limitations. We plan to utilize knowledge graphs to develop a holistic representation of data science artifacts and show the potential of linked data science by automating critical data science tasks.

# 6 Doctorate Work

Below is a summary of the academic achievements completed by Mossad Helali between January 2021 and June 2023 as part of the Ph.D. program at Concordia University.

## 6.1 Coursework

| Course | Name | Credits | Semester | Grade |
|---|---|---|---|---|
| COMP 6231 | Distributed System Design | 4.0 | Winter 2021 | A+ |
| COMP 6531 | Foundations/Semantic Web | 4.0 | Fall 2021 | A+ |
| SOEN 6111 | Big Data Analytics | 4.0 | Winter 2022 | A+ |

## 6.2 Publications

### 6.2.1 Journal Papers

- **M. Helali**, E. Mansour, I. Abdelaziz, J. Dolby, K. Srinivas, – "A Scalable AutoML Approach based on Graph Neural Networks", 2022, *Proceedings of the VLDB Endowment 15 (11), 2428-2436*, https://doi.org/10.14778/3551793.3551804

### 6.2.2 Demonstrations

- A. Helal, **M. Helali**, K. Ammar, E. Mansour – "A Demonstration of KGLac: A Data Discovery and Enrichment Platform for Data Science", 2021, Proceedings of the VLDB Endowment 14 (12), 2675-2678. https://doi.org/10.14778/3476311.3476317

### 6.2.3 Under Submission

- **M. Helali**, S. Vashisth, P. Carrier, K. Hose, E. Mansour, – “Linked Data Science Powered by Knowledge Graphs”, *Under Submission*, https://arxiv.org/abs/2303.02204

### 6.2.4 Unpublished Work

- W. Zhao, I. Abdelaziz, J. Dolby, K. Srinivas, **M. Helali**, E. Mansour, – “Serenity: Library Based Python Code Analysis for Code Completion and Automated Machine Learning”, https://arxiv.org/abs/2301.05108

## 6.3 Workshops

- The Second Data Systems Meet Data Science Workshop, Montreal, 2023. Poster Title: "Linked Data Science: Data Lakes Meet Pipeline Scripts".
- The First Data Systems Meet Data Science Workshop, Montreal, 2022. Poster Title: "A Scalable AutoML Approach Based on Graph Neural Networks".

# References


[1] Ibrahim Abdelaziz et al. “A Toolkit for Generating Code Knowledge Graphs”. In: *Proceedings of Knowledge Capture Conference (K-CAP)* (2021). URL: https://doi.org/10.1145/3460210.3493578.

[2] Ziawasch Abedjan and Felix Naumann. “Advancing the Discovery of Unique Column Combinations”. In: *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. CIKM ’11. Glasgow, Scotland, UK: Association for Computing Machinery, 2011, pp. 1565–1570. ISBN: 9781450307178. DOI: 10.1145/2063576.2063801. URL: https://doi.org/10.1145/2063576.2063801.

[3] Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. “DFD: Efficient Functional Dependency Discovery”. In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*. CIKM ’14. Shanghai, China: Association for Computing Machinery, 2014, pp. 949–958. ISBN: 9781450325981. DOI: 10.1145/2661829.2661884. URL: https://doi.org/10.1145/2661829.2661884.

[4] Ziawasch Abedjan et al. *Data Profiling*. Synthesis Lectures on Data Management. Morgan & Claypool, 2018. URL: https://doi.org/10.2200/S00878ED1V01Y201810DTM052.

[5] Miltiadis Allamanis et al. “A Survey of Machine Learning for Big Code and Naturalness”. In: *ACM Comput. Surv.* 51.4 (July 2018). ISSN: 0360-0300. DOI: 10.1145/3212695. URL: https://doi.org/10.1145/3212695.

[6] Mattia Atzeni and Maurizio Atzori. “CodeOntology: RDF-ization of Source Code”. In: *Proceedings of International Semantic Web Conference, (ISWC)*. Oct. 2017, pp. 20–28. ISBN: 978-3-319-68203-7. URL: https://doi.org/10.1007/978-3-319-68204-4_2.

[7] Sören Auer et al. “DBpedia: A Nucleus for a Web of Open Data”. In: *Proceedings of the 6th International The Semantic Web and 2nd Asian Conference on Asian Semantic Web Conference*. ISWC’07/ASWC’07. Busan, Korea: Springer-Verlag, 2007, pp. 722–735. ISBN: 3540762973.

[8] Alex Bogatu et al. “Dataset Discovery in Data Lakes”. In: *Proceedings of International Conference on Data Engineering (ICDE)*. 2020, pp. 709–720. URL: https://doi.org/10.1109/ICDE48307.2020.00067.

[9] Piotr Bojanowski et al. “Enriching Word Vectors with Subword Information”. In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 135–146. ISSN: 2307-387X.

[10] A.Z. Broder. “On the resemblance and containment of documents”. In: *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*. 1997, pp. 21–29. DOI: 10.1109/SEQUEN.1997.666900.

[11] Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. “Inclusion Dependencies and Their Interaction with Functional Dependencies”. In: PODS ’82. Association for Computing Machinery, 1982, pp. 171–176. ISBN: 0897910702. DOI: 10.1145/588111.588141. URL: https://doi.org/10.1145/588111.588141.

[12] Moses S. Charikar. “Similarity Estimation Techniques from Rounding Algorithms”. In: *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*. STOC ’02. Montreal, Quebec, Canada: Association for Computing Machinery, 2002, pp. 380–388. ISBN: 1581134959. DOI: 10.1145/509907.509965. URL: https://doi.org/10.1145/509907.509965.

[13] Ting Chen et al. “A Simple Framework for Contrastive Learning of Visual Representations”. In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119.

PMLR, 2020, pp. 1597–1607. URL: http://proceedings.mlr.press/v119/chen20j.html.

[14] E. F. Codd. “A relational model of data for large shared data banks”. In: *M.D. computing : computers in medical practice* 15 3 (1970), pp. 162–6.

[15] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, New York, 1999.

[16] Wei Dai, Kenji Yoshigoe, and William Parsley. “Improving Data Quality Through Deep Learning and Statistical Models”. In: *Information Technology - New Generations*. Cham: Springer International Publishing, 2018, pp. 515–522. ISBN: 978-3-319-54978-1.

[17] Xiang Deng et al. “TURL: Table Understanding through Representation Learning”. In: *Proc. VLDB Endow.* 14.3 (Nov. 2020), pp. 307–319. ISSN: 2150-8097. DOI: 10.14778/3430915.3430921. URL: https://doi.org/10.14778/3430915.3430921.

[18] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*. Association for Computational Linguistics, 2019, pp. 4171–4186. URL: https://doi.org/10.18653/v1/n19-1423.

[19] Grace Fan et al. *Semantics-aware Dataset Discovery from Data Lakes with Contextualized Column-based Representation Learning*. 2023. arXiv: 2210.01922.

[20] Raul Fernandez et al. “Aurum: A Data Discovery System”. In: *Proceedings of International Conference on Data Engineering (ICDE)*. 2018, pp. 1001–1012. URL: https://doi.org/10.1109/ICDE.2018.00094.

[21] Aidan Hogan et al. *Knowledge Graphs*. English. Synthesis Lectures on Data, Semantics, and Knowledge 22. Springer, 2021. ISBN: 9783031007903. DOI: 10.2200/S01125ED1V01Y202109DSK022. URL: https://kgbook.org/.

[22] Xin Huang et al. *TabTransformer: Tabular Data Modeling Using Contextual Embeddings*. 2020. arXiv: 2012.06678.

[23] Ykä Huhtala et al. "Tane: An Efficient Algorithm for Discovering Functional and Approximate Dependencies". In: *The Computer Journal* 42.2 (Jan. 1999), pp. 100–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/42.2.100. eprint: https://academic.oup.com/comjnl/article-pdf/42/2/100/986909/420100.pdf. URL: https://doi.org/10.1093/comjnl/42.2.100.

[24] Madelon Hulsebos et al. "Sherlock: A Deep Learning Approach to Semantic Data Type Detection". In: *ACM Knowledge Discovery and Data Mining (KDD)*. 2019. DOI: 10.1145/3292500.3330993. URL: http://vis.csail.mit.edu/pubs/sherlock.

[25] Yannis Ioannidis. "The History of Histograms (Abridged)". In: *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB) - Volume 29*. VLDB Endowment, 2003, pp. 19–30. ISBN: 0127224424.

[26] *Kaggle Portal. Accessed: 2022-07-15*. 2022. URL: https://www.kaggle.com/.

[27] Aamod Khatiwada et al. "SANTOS: Relationship-based Semantic Table Union Search". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2023.

[28] A. Koeller and E.A. Rundensteiner. "Discovery of high-dimensional inclusion dependencies". In: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. 2003, pp. 683–685. DOI: 10.1109/ICDE.2003.1260834.

[29] Christos Koutras et al. "SiMa: Effective and Efficient Data Silo Federation Using Graph Neural Networks". In: *ArXiv* abs/2206.12733 (2022).

[30] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. 2nd ed. Cambridge University Press, 2014. DOI: 10.1017/CBO9781139924801.

[31] Lu Liang et al. “KG4Py: A toolkit for generating Python knowledge graph and code semantic search”. In: *Connection Science* 34.1 (2022), pp. 1384–1400. DOI: 10.1080/09540091.2022.2072471.

[32] ZQ Lin et al. “Intelligent Development Environment and Software Knowledge Graph”. In: *Journal of Computer Science and Technology* 32.2 (2017), pp. 242–249. DOI: 10.1007/s11390-017-1718-y.

[33] Yu Malkov and Dmitry Yashunin. “Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2016). DOI: 10.1109/TPAMI.2018.2889473.

[34] Essam Mansour, Kavitha Srinivas, and Katja Hose. “Federated Data Science to Break Down Silos [Vision]”. In: *SIGMOD Rec.* 50.4 (Jan. 2022), pp. 16–22. ISSN: 0163-5808. DOI: 10.1145/3516431.3516435. URL: https://doi.org/10.1145/3516431.3516435.

[35] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. “Unary and n-ary inclusion dependency discovery in relational databases”. In: *Journal of Intelligent Information Systems* 32.1 (Feb. 2009), pp. 53–73. ISSN: 1573-7675. DOI: 10.1007/s10844-007-0048-x. URL: https://doi.org/10.1007/s10844-007-0048-x.

[36] meta-kaggle. *https://www.kaggle.com/kaggle/meta-kaggle*. 2022.

[37] Tomás Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: *1st International Conference on Learning Representations, ICLR*. 2013.

[38] Jonas Mueller and Alex Smola. “Recognizing Variables from Their Data via Deep Embeddings of Distributions”. In: *International Conference on Data Mining (ICDM)*. 2019, pp. 1264–1269. URL: https://doi.org/10.1109/ICDM.2019.00158.

[39] Fatemeh Nargesian et al. “Table Union Search on Open Data”. In: *Proceedings of the VLDB Endowment, (PVLDB)* (2018), pp. 813–825. URL: http://www.vldb.org/pvldb/vol11/p813-nargesian.pdf.

[40] Natasha Noy, Matthew Burgess, and Dan Brickley. “Google Dataset Search: Building a search engine for datasets in an open Web ecosystem”. In: *The Web Conference (WebConf)*. 2019.

[41] Thorsten Papenbrock and Felix Naumann. “A Hybrid Approach to Functional Dependency Discovery”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 821–833. ISBN: 9781450335317. DOI: 10.1145/2882903.2915203. URL: https://doi.org/10.1145/2882903.2915203.

[42] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian Suchanek. “YAGO 4: A Reason-able Knowledge Base”. In: *The Semantic Web*. Cham: Springer International Publishing, 2020, pp. 583–596. ISBN: 978-3-030-49461-2.

[43] Jeffrey Pennington, Richard Socher, and Christopher Manning. “GloVe: Global Vectors for Word Representation”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: https://doi.org/10.3115/v1/D14-1162.

[44] Vijayshankar Raman and Joseph M. Hellerstein. “Potter’s Wheel: An Interactive Data Cleaning System”. In: *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*. 2001, pp. 381–390. ISBN: 1558608044.

[45] Alexandra Rostin et al. “A Machine Learning Approach to Foreign Key Discovery.” In: Jan. 2009.

[46] Nuhad Shaabani and Christoph Meinel. “Detecting Maximum Inclusion Dependencies without Candidate Generation”. In: *Database and Expert*

*Systems Applications*. Cham: Springer International Publishing, 2016, pp. 118–133. ISBN: 978-3-319-44406-2.

[47] Yannis Sismanis et al. "GORDIAN: Efficient and Scalable Discovery of Composite Keys". In: *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 691–702.

[48] Beth M. Sundheim. "Overview of Results of the MUC-6 Evaluation". In: *Proceedings of the 6th Conference on Message Understanding*. MUC6 '95. Columbia, Maryland: Association for Computational Linguistics, 1995, pp. 13–31. ISBN: 1558604022. DOI: 10.3115/1072399.1072402. URL: https://doi.org/10.3115/1072399.1072402.

[49] Joaquin Vanschoren et al. "OpenML: Networked Science in Machine Learning". In: *SIGKDD Explorations* 15 (2013), pp. 49–60. URL: https://doi.org/10.1145/2641190.2641198.

[50] Rui Xie et al. "DeepLink: A Code Knowledge Graph Based Deep Learning Approach for Issue-Commit Link Recovery". In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2019, pp. 434–444. DOI: 10.1109/SANER.2019.8667969.

[51] Meihui Zhang et al. "Automatic Discovery of Attributes in Relational Databases". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. 2011, pp. 109–120. ISBN: 9781450306614. DOI: 10.1145/1989323.1989336. URL: https://doi.org/10.1145/1989323.1989336.

[52] Erkang Zhu, Yeye He, and Surajit Chaudhuri. "Auto-Join: Joining Tables by Leveraging Transformations". In: *Proc. VLDB Endow.* 10.10 (June 2017), pp. 1034–1045. ISSN: 2150-8097. DOI: 10.14778/3115404.3115409. URL: https://doi.org/10.14778/3115404.3115409.

[53] Erkang Zhu et al. “JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes”. In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD ’19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 847–864. ISBN: 9781450356435. DOI: 10.1145/3299869.3300065. URL: https://doi.org/10.1145/3299869.3300065.